Activity: Unit Tests and Test Driven Development

Task overview

This task is about using Unit Tests to apply the Test Driven Development methodology to implement an API that exposes to its clients two endpoints:

- `get_coordinates(IN city_name)`, that receives a city name and returns either an error (*not found* or any other error) or some representation of the latitude and longitude of the specified city.
- `get_distance(IN coordinates1, IN coordinates2)`, that receives two coordinates (represented in the same format as outputted by the previous function) and returns the distance between the two coordinates or some error.

Considerations

For the implementation of `get_coordinates(IN city_name)`, you are expected to use *openstreetmaps*. Specifically, you can query openstreetmaps using, for example, curl in the following way:

```
curl "https://nominatim.openstreetmap.org/search?q=lima,peru&format=json"
```

1 Tasks

1. Design the solution.
2. Write unit tests that checks structural aspects. For example:
   - `can_call_existing_endpoints_of_the_API` (a function that confirms that existing endpoints can be called)
   - `cannot_call_not_existing_endpoints_of_the_API` (a function that confirms that non-existing endpoints cannot cannot be called)
   - Similarly, functions that test that the functions one calls work with the correct number of parameters and do not work with the incorrect number.
3. Implement the necessary code to pass the unit tests defined in point (2).
4. Implement a new unit test called `endpoint_returns_something`, which checks that the endpoint is not only callable but also returns something.
5. Implement whatever is necessary in the code to pass the test written in point (4).
6. Write a new unit test called `the_result_is_correct_for_simple_cases`, which checks that the function returns the correct value for some specific input cases.
7. Write code to pass the unit test added in (6).
8. Write a test called `the_result_is_correct_for_all_inputs`, which tests that the matches the expected output for a vast set of possible inputs.
9. Write the code to pass test (8).

10. Measure the code coverage of the created code. It must be at least 98%. If it is less than 98%, add tests to improve it.

11. Perform stress tests and measure availability, latency, and the percentage of repeated code in the proposed implementation.

12. **Ensure that the code is maintainable.** Use strict naming conventions for variables, use indenters that do not allow merging code that has not yet passed through an indenter, and require long and explanatory comments before each function. Before each function, there must be comments that explain AT LEAST the following:

    - General description of the function's input/output.
    - Parameters of the function.
    - The idea of the function.

Note All the functions must be short and have a single responsibility. A function should not be a "sequence of things" but one thing. That thing can be something "high-level" and call other functions. But it must be clear "what each function does."

Continuation of the tasks (updated: June 3) and one additional tool: CODEOWNERS

CODEOWNERS is a technique that is named in this way precisely because its implementation consists of one single file named `CODEOWNERS`. This file specifies who is the owner of each one of the files in the repositories (the owner can be, for example, a *team* and, in this way, all the memebrs of the team will be considered owners). We can *require an approval of some CODE OWNER always*. This requirement helps to prevent that people change code that does not own. In other words: if you own a portion of the code, CODEOWNERS protects your code from being changed without your approval.

13. Generate documentation for your project. You can, for example, write the comments to your functions in a way that enables the usage of some automatic generator of documentation.

14. Design a Docker image that you will use as server, and deploy your API on it.

15. Add a CODEOWNERS file to your repository.

[Optional] Back to the tic-tac-toe...

After having done this mini-project and gained some familiarity with the tools, let's come back to the previous project of the tic-tac-toe. The goal now is:
Write a tic-tac-toe that works properly. You should make use of the TDD methodology that you have learnt and make sure that your code does not return invalid values. No player should be disqualified. After that, the second part of the problem is: *write an invencible tic-tac-toe player.*

1. Clone the `tic-tac-toe` repository, which is not written using TDD. However, you can (and you are expected to) use TDD in your specific part of the repo: your player.

2. Write many tests to make sure that your player does not return invalid moves.
3. Make sure your player passes the tests.
4. Run a test coverage of your code and make sure that you have tested it entirely.
5. Write comments to your code. The comments should follow the same style as in Sanders' solution to the API for the distance.
6. Create a PR. Ask 2 reviewers to approve it. Merge it.